CS168 Routing Simulator Guide (Fall 2015)

1 File Layout

Here are some highlights:
`simulator.py` - Starts up the simulator (see section below).
`learning_switch.py` - Starting point for your learning switch.
`dv_router.py` - Starting point for your distance vector router.
`sim/api.py` - Parts of the simulator that you'll need to use (such as the `Entity` class). See `help(api)`.
`sim/basics.py` - Basic simulator pieces built with the API. See `help(basics)`.
`sim/core.py` - Inner workings of the simulator. Keep out.
`topos/` - Test topologies and topology generators that you can use and modify for your own testing.
`examples/` - Examples for Entities, interacting with NetVis, automating your testing, and so forth.
`tools/` - Extra tools

2 Understanding the Simulator Commandline

The typical way to start the simulator is using `simulator.py`. This starts up a number of "modules" and then begins simulation. Modules are Python scripts which typically implement a `launch` function. A module might, for example, create a topology or run an automated test.

`simulator.py` itself may take commandline options. These come at the very beginning of the commandline and are prefixed with two dashes (e.g., `--foo`). Commandline arguments without the two dashes specify the names of modules to be loaded (in the order in which they'll be loaded). Modules can also have options (again prefixed with two dashes). So a complete commandline might look like:

```
$ python simulator.py --sim-opt1=value --no-sim-opt2 mod1 --mod1-opt1=value mod2 --mod2-opt1
```

Here we're passing two options to the simulator, and loading two modules, each of which are getting one option. Note that normally options are of the form `--name=value`. For boolean options, you just do `--name` or `--no-name`.

Module options are automatically passed to the module's `launch` function.

3 Micro-Tutorial

Let's try a quick introduction. Since you haven't implemented a router yet, we'll use the simple example Hub entity which just takes every packet it receives and sends it on every other link.

We'll use the `topos.rand` module to create a random topology with four switches/routers (hubs in this case), four hosts (one per switch), and three links. Since there are four switches and only three links, this topology is guaranteed not to have loops, which is important because hubs don't do so well on topologies with loops!

```
$ python simulator.py --default-switch-type=examples.hub topos.rand --switches=4 --links=3 \
        --hosts=4 --no-multiple-hosts
```

Executing the above should give some informational text (including a listing of the names of each node in the

topology – h1-h4 and s1-s4 in this case) and then a Python interpreter in the terminal. If you want to try to the NetVis GUI (see the section later), you can start it now. If the simulator doesn't start up, the most common reason results in a traceback for an `Address already in use` exception. If this occurs, it's likely that someone else is running the simulator on the same machine (or you have an old instance running which you should quit!). This is going to make using the log viewer and NetVis (described in later sections) impossible without hacking them a little, which may or may not be a problem depending on whether you want to use them. For now, add the `--no-remote-interface` option.

From the interpreter in the terminal, you can run arbitrary Python code, can inspect and manipulate the simulation, can interact with the Entities (switches, hosts, etc.), and can get help on many aspects of the simulator. For example `help(topos.rand)` will show information on the random topology generator, and `help(h1)` will give help on the first host node.

Let's start the simulation (it's initially in a suspended state unless you pass `--start` on the commandline) and send a ping between two of the hosts:

```
>>> start() # make sure not to forget this!
>>> h1.ping(h2)
```

(If you're using NetVis, you could also have sent the ping by selecting `h1`, pressing "a", selecting `h2`, pressing "b", and then pressing "p".)

After a short wait, the ping packets will start hitting the other hosts; when this happens, a log message is printed. Since we're using a hub, the packet will show up at *all* other hosts – when it shows up at the wrong one, the log message will say so. Hosts, by default, respond to pings by sending back a pong. These will also show up in the console.

So why don't we want to use a bunch of hubs on a topology with loops? Because the packets just keep looping around. You can try it yourself – press Ctrl-D (or type `exit()`) to exit the simulator, and then re-run it passing `--links=4` which ensures there's a loop. Try the ping test again. When you get bored, exit the simulator again. Note that the simulator automatically gives us a TTL – any packet will only be forwarded up to some maximum number of times – or the amount of looping and packet replication would be even greater!

By modifying the commandline above to use `learning_switch` rather than `examples.hub`, you can get started working on your own learning switch.

4 Implementing Your Own Distance Vector Router

When you've mastered your learning switch, you can get started on your distance vector router. First things first, let's start by changing the commandline we'll use for the simulator. Rather than using a random topology, let's start with something very simple – like a simple linear topology. We can use the `topos.linear` topology generator for that. As your implementation matures, you can try some of the other topologies we've included, or make your own with a Python script or using the `.topo` file format loaded by `topos.loader`. We also want to make sure to use your `DVRouter` class instead of the hub – we've provided a skeleton of this class in `dv_router.py`, which you can modify. So our first commandline might look like:

```
python simulator.py --default-switch-type=dv_router topos.linear
```

Once you've got that going, it's up to you to implement distance vector routing, by sending routes to neighbors, processing the ones they send you, and listening for hosts to announce themselves with their discovery messages. See the assignment documentation for more on all of this, including things your router will be expected to do (and not do), specific packet types you'll have to deal with, and other tips. The remainder of this section also touches on a few details.

4.1 Implementing Entities

Objects that exist in the simulator are subclasses of the Entity superclass (in `api.py`). These have a handful of utility functions, as well as some empty functions that are called when various events occur. You probably want to handle at least some of these events! For more help try `help(api.Entity)` within the simulator. You might also want to peek at `hub.py`, which is a simple `Entity` to get you started.

Your `DVRouter` should actually subclass a subclass of `Entity` – the DVRouterBase class. Basically this subclass just adds support for timed updates and a flag for whether the router should send poison routes or not. For reference, some important functions from `Entity` are as follows (feel free to read the documentation/comments in `dv_router.py`, poke around in the source or try `help(api.Entity)` and `help(basics.DVRouterBase)` in the simulator for more information):

```
class Entity (object)
    handle_rx (self, packet, port)
        Called by the framework when the Entity self receives a packet.
        packet - a Packet (or subclass).
        port - port number it arrived on.
        You definitely want to override this method.

    handle_link_up (self, port, latency)
        Called by the framework when a link is attached to this Entity.
        port - local port number associated with the link
        latency - the latency of the attached link
        You probably want to override this method.

    handle_link_down (self, port)
        Called by the framework when a link is unattached from this Entity.
        port - local port number associated with the link
        You probably want to override this method.

    send (self, packet, port=None, flood=False)
        Sends the packet out of a specific port or ports. If the packet's
        src is None, it will be set automatically to the Entity self.
        packet - a Packet (or subclass).
        port - a numeric port number, or a list of port numbers.
        flood - If True, the meaning of port is reversed - packets will
        be sent from all ports EXCEPT those listed.
        Do not override this method.

    log (self, format, *args)
        Produces a log message
        format - The log message as a Python format string
        args - Arguments for the format string
        Do not override this method.
```

4.2 Sending Packets

Entities can send and receive packets. In the simulator, this means a subclass of Packet. See `basics.Ping` for one such example, and see `basics.BasicHost` to see an example of how it is used. You can tell packet types apart using Python's `isinstance()`, e.g., `if isinstance(packet, basics.Ping): ...`

Here are some highlights of the `Packet` superclass (as usual, `help(api.Packet)` will tell you more):

```
class Packet (object)
    self.src
    Packets have a source address.
    You generally don't need to set it yourself.  The "address" is actually a
    reference to the sending Entity, though you shouldn't access its attributes!

    self.dst
    Packets have a destination address.
    In some cases, packets aren't routeable -- they aren't supposed to be
    forwarded by one router to another.  These don't need destination addresses
    and have the address set to None.  Otherwise, this is a reference to a
    destination Entity.

    self.trace
    A list of every Entity that has handled the packet previously. This is
    here to help you debug. Don't use this information in your router logic.
```

Subclasses may add more attributes. Aside from the aforementioned `Ping` subclass, you'll also be very interested in the `RoutePacket` and `HostDiscoveryPacket` which are discussed in a bit more detail in the assignment document.

4.3 Other Useful Stuff

Other things which may be useful:

1. `api.current_time()` returns the current time in seconds. This is useful to know when to time out routes, since you can't really be sure how often `handle_timer` will be called.
2. *Packets have colors* in the GUI. Pings default to random color outsides with whiteish inside. Pongs flip those so you can sort of see which pong went with which ping. Host discovery packets are yellow. Routes are magenta. If you want, you can control packet colors using the `inner_color` or `outer_color` attributes. Set them to a list containing `[redness, greenness, blueness, opacity]` where all values are between `0` and `1`.

5 Experimenting with Topologies

The simulator comes with a few different topologies and topology generators in the `topos` package. You can modify these, write your own based on them, etc. The point is, your router should work on arbitrary topologies, not just the ones we give you, so you might want to build your own to test on and experiment with! Here we briefly cover how you can go about this (besides just reading the code for the existing ones, which isn't that complex on the whole).

There are two basic ways of creating a topology. The first is programmatically, by creating and linking entities. The second is by creating a `.topo` topology file which is loaded by `topos.loader` (which, internally, just does things the first way).

5.1 Programmatic Topologies

The first step is simply creating some entities so that the simulator can use them. You shouldn't create the nodes using the normal Python instance creation. Instead, use the `.create()` factory on `Entity`, along the

lines of:
```
>>> MyNodeType.create('myNodeName')
```
That is, you **don't** want to use normal Python object creation like:
```
>>> x = MyNodeType() # Don't do this
```
You can use the return value from `create` if you want, though (it returns the new entity). You can also pass extra values into `create`, and they get passed to the constructor (`__init__`), but be careful with this feature, since we will initialize your routers with no additional arguments!

New entities should be globally available, so you can do:
```
>>> x = MyNodeType.create('myNodeName')
>>> print myNodeName, x
```
which will show the new `Entity` twice.

To link this to some other Entity:
```
>>> myNodeName.linkTo(someOtherNode)
```

You can also `.unlinkTo()` it, or disconnect it from everything:
```
>>> myNodeName.disconnect()
```

Note that you can use this same functionality to experiment with and interactively test your code live in the simulator.

5.2 Topo Files

There's a simple topology file format which is read by `topos.loader`.

The file format consists of lines beginning with `h` (for hosts), `s` (for switches), or `l` (links). Host and switch lines specify a node name. Link lines specify the two nodes to link and optionally a link latency.

You load these by using the `topos.loader` module with the `--filename` option on the commandline, like so:

```
$ python simulator.py topos.loader --filename=foo.topo
```

An example topology file looks like this (check the topos directory for more):

```
# A triangular thing
h A
h B
h C
s s1
s s2
s s3
l A s1
l B s2
l C s3
l s1 s2 0.5
l s2 s3 0.5
l s3 s1
```

6 Building Your Own Tests

It will probably help to test your code. You can do this interactively using the Python console (using the programmatic topology manipulation features mentioned above) or through the NetVis GUI. Another option is to write youself some test scripts to automate the testing process. We include a couple test modules to get you started in the `examples` directory.

The first of these is `test_simple.py`, which just creates a small topology and sends a couple pings, making sure the right number of pings arrive. Let's try it using the hub:

```
$ python simulator.py --no-interactive --default-switch-type=examples.hub examples.test_simple
```

Note that we use `--no-interactive` to start the simulator without the Python console for the purposes of testing.

Running this test with the hub should fail because packets show up where they shouldn't. Your distance vector router should pass it!

The `test_failure` test can be run similarly. It sends a packet, fails a link, sends another packet, and expects both packets to arrive at the destination. It'll also fail with the hub.

Using the code and design of these tests for inspiration, you can construct your own.

7 The Log Viewers

Log messages are generally sent to the terminal from which the simulator is run. If you are using the interactive interpreter for debugging or experimentation, this can be somewhat irritating. You can disable these by passing `--no-console-log` on the simulator commandline. Note that this needs to be *before* you list any modules, or this will be interpreted as an option for the preceeding module (as discussed in the section on the commandline above).

Of course, those log messages can be really quite helpful, and you might want to see them. To rectify this, the simulator comes with two standalone log viewers: `tools/logviewer.py` and `tools/console_logviewer.py`. The former is a GUI application which opens a new window, and you can run it in the background before starting the simulator:

```
$ python tools/logviewer.py &
```

The latter is a plain old console application, which you can run in a separate console (e.g., a new Terminal window on a Mac or a new ssh session if you're running the simulator on a remote machine).

8 NetVis: The GUI

The simulator includes a graphical user interface called NetVis. *This is mostly unsupported*, and you definitely don't need to use it in order to successfully complete the project. But some students have found it useful, and you're welcome to check it out.

NetVis is a visualization and interactivity tool for the network simulator. It is written using Processing, which is basically a framework for writing visualization tools in Java (with some nice shortcuts). You should have gotten the NetVis code (which is a mess these days). To get it running you can:

1. Download Processing from `processing.org`). It definitely works with the latest stable version (2.2.1 as

of now). It also seems to work with the latest beta of Processing 3.0.

2. Download the controlP5 library from `www.sojamo.de/libraries/controlP5`. Follow the installation instructions that come with it. In short, these say to find Processing's "libraries" directory (you can probably find this inside its sketchbooks directory, which you can probably find via the preferences/settings dialog), and then put controlP5 into it.

3. Possibly install Java.

You may also just be able to find NetVis executables posted on Piazza.

Entities, links, and packets in your simulator should show up automatically in NetVis. Generally these try to position themselves automatically. You can also "pin" them in place, by clicking the attached bubble. You can zoom in and out, as well as altering some of the layout forces using options in the "layout" panel (click on it to open it).

The "info" panel can be set from within the simulator, so you can populate it with helpful debugging info or whatever you like. If you open it up, it should start out with some usage tips.

See `examples/megaping.py` for an example of some advanced NetVis functionality.